Gradaive Matthias
Chiquet Romain

Groupe 2.1 GL

Compte rendu TP Supermaché

Le but de ce TP était de simuler le fonctionnement d'un supermarché. Dans ce supermarché on peut distinguer différents: acteurs les clients, l'employé de caisse ainsi que le chef de rayon. Ces différents acteurs interagissent avec des éléments du supermarché. Ces éléments sont les chariots, les rayons qui contiennent les différent articles vendus ainsi que la caisse.

Présentation des classes du projet:

Dans ce projet nous représenteront les différents acteurs et éléments du supermarché dans des classes. Les méthodes de ces classes représentent les actions réalisables par les acteurs.
Afin de simuler le fonctionnement d'un supermarché, certaines classes seront des threads et d'autres assureront la synchronisation entre ces threads.

Les objets qui sont des Threads sont les instances des classes Client, ChefRayon, Employe

Classe Supermarche: contient toutes les constantes relatives au fonctionnement du supermarché (nombre de rayon, nombre de chariot). Cette classe initialise les différents objets du supermarché.

Classe Client: contient les informations relatives au client un identifiant ainsi qu'une liste de course générée aléatoirement. Les instances de cette classe sont des **Threads**.

Classe ChefRayon: contient les informations relatives au chef de rayon, une liste de rayon qu'il aura besoin de parcourir. Les instances de cette classe sont des **Threads**. Ce Thread est plus précisément définit comme étant un daemon (il s'arrêtera quand tous les autres threads seront terminés).

Classe Employe: contient les informations relatives a l'employé de caisse, ainsi que le déroulement d'un cycle pour un employé de caisse. Cette classe a un accès à la classe Caisse. Les instances de cette classe sont des **Threads**.

Classe FileChariot: Cette classe représente les chariot qui sont utilisés par les clients, elle représente un **objet partagé**. Chaque client prend un chariot avant d'aller faire ses courses.

Classe Rayon: Cette classe représente un **objet partagé**. L'objet rayon, a besoin d'être accédé par les instances de Client et de ChefRayon.

Classe Caisse: Cette classe représente un **objet partagé.** Cette classe représente le tapis de caisse. Cet objet est utilisé par les clients ainsi que par l'employé de caisse.

Les différents motifs d'exclusions et leurs résolutions :

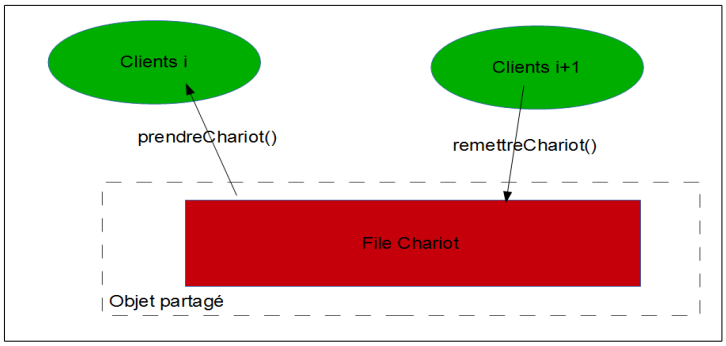
Du faite de la concurrence entre les différents Threads des motifs d'exclusions apparaissent.

La file de chariot et les clients:

La problématique :
Un certain nombre de chariot est disponible dans le supermarché. Si le nombre de client est supérieur à ce nombre de chariots alors il peut y avoir un problème de disponibilité.

Lorsqu'un client essaye de prendre un chariot, il faut vérifier si il y en a suffisamment, dans le cas présent au moins un.

Schéma de l'exclusion mutuelle sur les chariots



La résolution :
Ce type de problématique se rapproche du TP2, le système de location de vélo.
Pour résoudre ce problème nous avons deux méthodes d'accès aux chariots (qui sont représenté par un entier) afin de garantir l'exclusion mutuelle.

La prise d'un chariot ne peut pas être interrompue, il en va de même pour la remise.
La classe FileChariot fourni deux méthodes d'accès aux chariots, prendreChariot() et remettreChariot(). Ces deux méthodes sont synchronized.

Le client test il est possible de prendre un chariot, si le nombre de chariot est suffisant on fourni un chariot au client (on décremente l'entier nbChariot).
Si il n'y a pas suffisamment de chariot, le Thread représentant le client se met en attente (méthode wait()), c'est de l'attente passif.

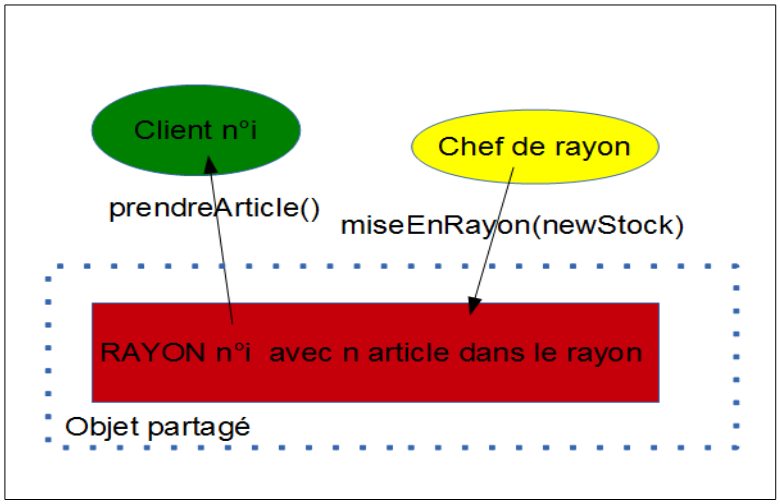
Lorsqu'un client remet un chariot (on incremente nbChariot), il notifie les clients qui pouvaient être en attente (méthode notifyAll()).

Les rayons, les clients et le Chef De Rayon:

La problématique :
Les clients du supermarché se déplacent de rayon en rayon afin de prendre le nombre d'article indiqué dans leur liste. Pour que le client puisse prendre le bon nombre d'article, il faut que ceux-ci soient disponibles (problématique semblable à la file de chariot avec les clients). Deux clients ne peuvent pas prendre le même article en même temps.
Si un article n'est pas disponible le client se met en attente, le chef de rayon a pour mission de recharger le contenu des rayons.

Le chef de rayon tout comme les clients doit agir sur le contenu des rayons. Ce problème est très semblable au Camion dans le système de location de vélo. Il doit recharger leurs contenus. Le chef de rayon ne peut pas agir sur le même article d'un rayon en même temps qu'un client (prise de produit et mise en rayon sont mutuellement exclus).

Schéma de l'exclusion mutuelle sur le rayon



La résolution :
Ce problème d'exclusion mutuelle entre les clients, le chef de rayon et les rayons est semblable à celui de la file de chariot et des clients.

Lorsqu'un client souhaite prendre un article, on teste si le stock du rayon est suffisant. Si le stock est insuffisant le client se met en attente (wait()). Lorsque le chef de rayon a rechargé le contenu d'un rayon, il prévient les clients en attente (notifyAll()).

Le chef de rayon continue de parcourir les rayons et vérifie si le stock de chacun est suffisant. Si besoin est il recharge.
Une fois que le chef de rayon a parcouru l'ensemble des rayons il recharge l'ensemble de son chariot à l'entrepôt.
Le chef de rayon est représenté par un Thread Daemon. Il s'execute en arrière plan, le thread meurt lorsque tous les autres threads ont terminé de s'executer.

Les méthodes d'accès à l'objet partagé sont miseEnRayon() et priseArticle(), elles sont syncronized.

Les clients et la caisse:

Problématique :
Il ne peut y avoir qu'un seul client à un moment donné qui dépose ses articles sur le tapis.
Le tapis est représenté par un buffer circulaire. Le client et l'employé doivent se synchroniser avant d'effectuer le paiement (le client attend que l'employé rencontre le marquer CLIENT_SUIVANT).

Résolution :

On utilise un sémaphore binaire (mutex), afin de n'autoriser qu'un seul client à la fois. La prise du sémaphore se fait lors de l'arrivée en caisse. La libération se fait après le paiement.

Le fonctionnement du tapis est à rapprocher du problème producteur/consommateur. Ici le client est le producteur tandis que l'employé est le consommateur.
Le dépôt et la prise de produit sur le tapis sont mutuellement exclus afin d'assurer le bon fonctionnement du buffer circulaire.
La gestion du tapis est assuré avec trois variables : l'index de dépôt (index d'écriture), l'index de prise (index de lecture) et le nombre total de produit sur la tapis (pour ne pas écraser une case du buffer pas encore lue).

Lors de l'arrivée en caisse d'un nouveau client, un nouveau Thread employé est créé (il y aura donc un thread par client). L'employé va prendre les produits présents sur le tapis un par un, ou attendre (wait()) si le tapis est vide. Le thread Employe se termine lors de la rencontre du marqueur CLIENT_SUIVANT.

Après la prise du sémaphore, le client commence à déposer un par un ses produits sur le tapis (avec une temporisation entre chaque dêpot). Il termine en déposant le marqueur CLIENT_SUIVANT.
Si le tapis est plein, le client attend (wait()) qu'une place se libère (notify réalisé quand l'employé prend un produit).

Il attend ensuite que l'employé de caisse terminé de passer tous les articles. Pour cela, on utilise un join() sur le thread de l'employé depuis le thread du client.

On peut ensuite réaliser le paiement puis libérer la caisse en libérant le mutex.
Le client peut ensuite déposer son chariot et terminer l'exécution de son thread.